Vol.2013-HPC-138 No.20
2013/2/22

協調マルチタスキングを用いて短い遊休時間を活用するGPUグリッドシステムの提案

岡 陽介[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，ミリ秒単位の遊休時間を活用することを目的として，GPU（Graphics Processing Unit）向けの協調マルチタスキングを用いたグリッドシステムを提案する．GPU グリッドは，ネットワーク上の共有計算資源として遊休 GPU を用い，大規模計算を加速する．既存システムは，遊休資源を検出するために，マウス・キーボード入力を監視し，CPU および GPU の計算負荷と組み合わせる手法を提案している．しかし，1 秒程度の短い遊休時間を活用できない問題がある．そこで提案システムは，協調マルチタスキングを用いて GPU プログラムを実行し，ミリ秒単位の短い遊休時間を活用する．提案システムは計算タスクを分割し，GPU の負荷に応じて分割数および実行モードを切り替えながら，各々を実行する．これらにより，描画処理および GPU プログラムの並行実行を実現し，資源所有者に与える外乱を抑える．実験では，研究室の学生が日常的に使用する 4 台の計算機に対して提案システムを運用した．結果，既存システムと比較して最大で 1.7 倍の遊休時間を検出し，フレームレートを低下することなく行列積のスループットを倍増できた．

キーワード：グリッド計算，協調マルチタスキング，GPU，ボランティア計算

A GPU-Accelerated Grid System for Exploiting Short Idle Time with Cooperative Multitasking

Yosuke Oka[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: This paper proposes a grid system based on a cooperative multitasking technique for the graphics processing unit (GPU), aiming at exploiting short idle time in the order of milliseconds. GPU grids accelerate large-scale computation by exploiting idle GPUs as shared computational resources on the network. To detect idle resources, a previous system monitors mouse and keyboard activities, which are then combined with CPU and GPU load averages. However, this approach cannot exploit short idle time such as a second. To exploits such short idle time of milliseconds, our proposed system executes GPU programs with a cooperative multitasking technique. Our system divides a computational task into smaller parts, which are then executed with the appropriate number of divisions and execution mode determined according to GPU workload. Owing to this technique, we not only realize concurrent execution of frame updates and GPU programs but also minimize the perturbation to the resource owners. In experiments, we apply our system to four machines ordinarily used by students in our laboratory. As a result, our system detects 1.7 times longer idle time than a previous system, and doubles the performance of matrix multiplication without dropping the frame rate.

Keywords: Grid computing, cooperative multitasking, GPU, volunteer computing

1. はじめに

GPU はグラフィクス処理用のアクセラレータである．

[1] 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

GPU は，CPU に比べて高い浮動小数点演算性能を持つ．NVIDIA 社が開発環境 CUDA （Compute Unified Device Architecture）[1] を公開したことにより，GPU プログラムの開発は容易になりつつある．結果，GPU の高い演算性能に着目し，汎用計算への応用を試みる GPGPU（General Purpose computation on the GPU）の研究が盛んである．

その研究の 1 つとして，GPU を用いてグリッドシステムを高速化する試みがある．ここで，本稿におけるグリッドシステムとは，ネットワーク上の計算資源を共有し，科学計算を高速化するボランティア型の計算システムを指す．以降では，計算資源を提供する側をホストと呼び，提供された計算資源を用いて高速化する側をゲストと呼ぶ．また，高速化の対象となる科学計算をゲストタスクと呼び，計算資源上でローカルに動作する描画処理などをホストタスクと呼ぶ．例えば，2 万台の GPU を用いる Folding@home プロジェクト [2] では，ゲストタスクとしてタンパク質の折り畳みシミュレーションを高速化している．

GPU を用いるグリッドシステム（GPU グリッド）では，同一 GPU 上においてホストタスクおよびゲストタスクが同時に実行されうる．この際，単純にゲストタスクを実行してしまうと，ホストタスクの実行が妨げられてしまい，フレームレート（1 秒当たりの画面更新回数）が低下する．この低下はゲストタスクの実行時間が長くなるにつれ顕著になり，1 fps（frames per second）を下回ることもある．フレームレートが低下する理由は，現在の GPU がハードウェアによるプリエンプティブなマルチタスキング機構を持たないことにある．したがって，ホストに対する外乱を最小化しつつ，ゲストに提供する性能を最大化する必要がある．

そこで，Folding@home プロジェクト [2] では，スクリーンセーバを用いて遊休ホストを検出し，それら計算資源の状態を資源管理サーバに集めて管理している．これによりホストタスクおよびゲストタスクが同時に実行されることを回避できる．ただし，この検出方法はスクリーンセーバのタイムアウト時間に頼っていて，遊休状態が少なくとも数分程度に渡り持続することを前提にしている．結果，数秒程度の遊休時間を活用することはできない．

一方，数秒程度の遊休時間を活用することを目的として，既存研究 [3] ではマウス・キーボードの入力を監視し，CPU および GPU の計算負荷と組み合わせる検出方法を採用している．さらに，外乱を最小化するために，ゲストタスクの実行を 1 回あたり 100 ミリ秒以内に終えるように，ゲストタスクを分割している．しかし，ホストがマウス・キーボードを操作していたとしても，必ずしも GPU が繁忙であるとは限らず，ゲストに計算資源として提供できる余地がある．また，マウス・キーボード入力は 1 秒程度の短い間隔で検出されることが多い．したがって，ゲストタスクの割り当て直後に遊休状態から繁忙状態に変わることが原因で，タスク実行に失敗することが多い．さらに，秒単位の遊休時間を対象としたことが原因で，スクリーンセーバによる検出方法と比べて GPU の状態遷移が頻繁である．したがって，資源管理サーバへの通知が多発しサーバを圧迫する．

本研究では，これらの問題を解決するために，GPU 向けの協調マルチタスキング [4] を用い，ミリ秒単位の短い遊休時間を活用する GPU グリッドを提案する．既存研究 [3] と同様に，協調マルチタスキングはゲストタスクを分割して実行する．ただし，2 つの実行モードを備えることにより，フレームレートを低下させることなく，マウス・キーボードを入力しながらゲストタスクを並行実行する．これによりマウス・キーボード入力を監視することなく，外乱の最小化を図る．これにより，入力の検出に起因する状態遷移回数を削減でき，サーバの圧迫を軽減できる．

以降では，まず 2 節で協調マルチタスキングについて述べる．次に，3 節で提案システムを示し，4 節で評価実験の結果を示す．最後に，5 節で本稿をまとめる．

2. 協調マルチタスキング

提案システムの協調マルチタスキング [4] は，ホストのフレームレートを維持しつつ，ゲストタスクのスループットを最大化する．これらはトレードオフの関係にある．したがって，よいトレードオフポイントを探索し，両者の程度を制御できる仕組みが必要である．協調マルチタスキングでは，この仕組みを以下の 3 点により実現する．

(1) ゲストタスクの分割：分割により外乱の最小化を図る．以降では，分割済のゲストタスクの各々をサブタスクと呼ぶ．
(2) GPU 負荷の推定：GPU の負荷に応じてサブタスクの実行モードを切り替えることによりスループットの向上を図る．
(3) 実行モードの選択：GPU の負荷が低い場合，サブタスクを連続実行する．そうでない場合，サブタスクを一定の時間間隔で実行しホストタスクに実行機会を与える．

以降，各々について説明する．

2.1 ゲストタスクの分割

CUDA は，異なるスレッドブロック（TB）がデータ依存に関して互いに独立であることを前提にしている．そこで，この独立性に着目し，ゲストタスクを TB 単位で分割する．そのために必要な修正は，以下の 3 点である．

- カーネル関数に与える TB 数の削減：カーネル関数の引数を修正し，カーネル呼び出し 1 回あたりの TB 数を削減する．
- ループ文の追加：上記の TB 数を削減した結果，カーネルの呼び出し回数を増やす必要がある．そのための

ループ文を CPU コードに追加する．

- 出力番地の補正：各 TB が出力する番地のオフセットをカーネル関数の引数として渡し，GPU コード内で出力番地を補正する必要がある．

TB の適切な分割数は，カーネル呼び出し 1 回あたりの実行時間を基に決める．そのための前提として，本研究ではホストタスクが一定の時間間隔 $1/F$ ごとに繰り返し実行されるものとする．ここで，F はフレームレートを表す．ホストタスクの実行時間を 1 回あたり T_r とすれば，サブタスクの実行時間 T_s が次式を満たせば，時間間隔 $1/F$ は遅延しない．

$$T_s \leq 1/F - T_r \quad (1)$$

したがって，式 (1) を満たすようにゲストタスクを分割すればよい．一般に，ゲストタスクの分割数を増やしすぎると，実行効率が低下してしまうため，T_s は大きな値の方がよい．ゆえに，式 (1) を満たし，ゲストタスクのスループットを最大化する値を実験的に求める．

2.2 GPU 負荷の推定

現在の GPU ドライバは GPU の計算負荷を提供している．しかし，その時間分解能は低く，ミリ秒程度の間隔で更新される情報を取得することは不可能である．また，プリエンプションを提供しない GPU 上では，計算負荷を検出することそのものがフレームレートを低下させる恐れがある．

そこで，フレームレートを可能な限り維持しつつ GPU 負荷を推定するために，提案システムはサブタスクを呼び出す直前に空カーネルの実行時間 T_d を測定する．ここで，空カーネルとは呼び出した直後に呼び出し元に制御を戻すデバイス関数である．あらかじめ遊休状態のときに計測しておいた空カーネルの実行時間 σ を T_d と比較し，負荷の程度を推定する．具体的には，$\sigma \leq T_d$ の場合は高負荷とみなし，$\sigma > T_d$ の場合は低負荷とみなす．

この推定方法は誤検出の問題がある．例えば，低負荷であると誤って推定した場合，ゲストタスクの過度な実行によりフレームレートを維持できない．一方，高負荷であると誤って推定した場合，ゲストタスクのスループットは最大化できないものの，フレームレートを維持できる．提案システムでは，ホストへの外乱を抑えることを優先し，後者は問題とみなさない．前者に対しては，その誤検出を避けるために，R 回ほど連続して $\sigma > T_d$ を満たす場合にのみ低負荷状態であると判定する．なお，T_s と同様に，試行回数 R の値は実験的に定める．

2.3 実行モードの選択

提案システムは，GPU の負荷に応じて 2 つの実行モードを切り替えてサブタスクを実行する．GPU の負荷が高い場合は定期実行モードを用い，低い場合は連続実行モードを用いる．**図 1** に，切り替えの基となる状態遷移図を示す．

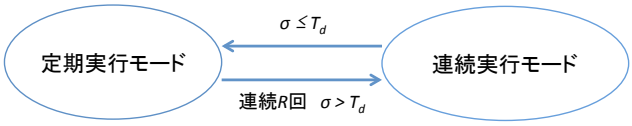


図 1 実行モードの切り替え

フレームレートを維持するために，定期実行モードはサブタスクを実行するたびに $1/F$ の待ち時間を設ける（**図 2(a)**）．これにより，ホストタスクに少なくとも $1/F$ の周期で実行機会を与えることができ，フレームレートの低下を回避できる．仮に，式 (1) を満たすように分割したゲストタスクを待ち時間なしで実行する場合，必ずしもゲストタスクおよびホストタスクが交互に実行されるとは限らない．結果，ゲストタスクが連続して GPU を専有してしまい，それらの実行時間が $T_r - 1/F$ を超えることでフレームレートが低下してしまう．なお，$1/F$ の待ちは，サブタスクの実行後に待ち関数を呼び出すことで実現する．待ち関数には Windows API [5] の Sleep 関数を用いる．

一方，連続実行モードにおいては，ゲストタスクのスループットを高めるために，待ち時間なしでサブタスクを連続実行する（**図 2(b)**）．GPU の負荷が低い場合，ホストタスクの実行頻度は低いため，負荷が高いときよりも GPU の空き時間は長い．したがって，この空き時間にサブタスクを可能な限り連続実行することにより，定期実行モードよりもスループットを向上できる．

図 3 に，ゲストタスクを実行するための制御アルゴリズムを示す．このアルゴリズムの入力は，ゲストタスク，保証したいフレームレート F，空カーネルの実行時間 σ，試行回数 R，タスク分割後の TB 数 N_1 および N_2 である．ここで，N_1 および N_2 は 2.1 節で述べたようにサブタスクの実行時間 T_s を基に定める．なお，ゲストタスクのスループットを高めるために，連続実行モードは定期実行モードよりも多くの TB を割り当てる．つまり，$N_1 < N_2$ である．

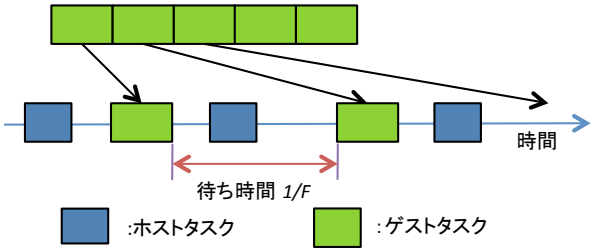
3. 提案システム

本節では，提案する GPU グリッドシステムの概要について述べる．

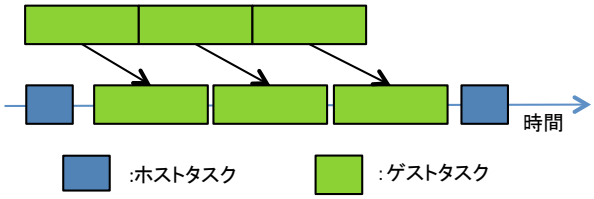
3.1 計算資源の状態定義

本研究では，以下に示す 2 つ条件を満たす計算資源を繁忙資源と定義する．すなわち，以下の条件を満たさない資源が遊休状態である．

条件 1. CPU 使用率が $x\%$ 以上であること．ゲストタスクは GPU サイクルのみならず，CPU サイクルも消費する．特に，最新の Kepler アーキテクチャでは，GPU



(a) 定期実行モード



(b) 連続実行モード

図 2 GPU の計算負荷に応じた実行モード

入力： ゲストタスク，閾値 σ，試行回数 R，フレームレート F，
定期実行モードのための TB 数 N_1，
連続実行モードのための TB 数 N_2．

```
1: count := 0; T_d := 0;
2: while ゲストタスクが終了していない do
3:   if σ > T_d then
4:     count := count + 1;
5:   else
6:     count := 0;
7:   end
8:   T_begin := 時刻;
9:   if count < R then // 高負荷：定期実行モード
10:    TB 数 N_1 でカーネルを実行;
11:    Sleep(1/F 秒);
12:  else // 低負荷：連続実行モード
13:    TB 数 N_2 でカーネルを実行;
14:  end
15:  T_d := 現在時刻 −T_begin;
16: end
```

図 3 ゲストタスクの実行制御アルゴリズム

内部のタスクスケジューリングの一部をドライバが担当していて，CPU の計算負荷は高くなる傾向がある．したがって，CPU 使用率が高い計算資源にゲストタスクを投入すべきでない．なお，ホストが許容できる外乱を指定できるように，x の値はパラメータとして設定できるようにしている．デフォルトの値は $x = 30$ である．

条件 2. ホスト起動直後の状態から新たに y MB 以上のビデオメモリ領域を確保していること．この条件の目的はホストが対話的に GPU を使用していないにも関わらず，GPU が繁忙なケースを検出することである．例えば，CUDA プログラムをホストタスクとして実行している場合が当てはまる．条件 1 と同様に，y の値はパラメータとして設定でき，デフォルトの値は $y = 1$ である．

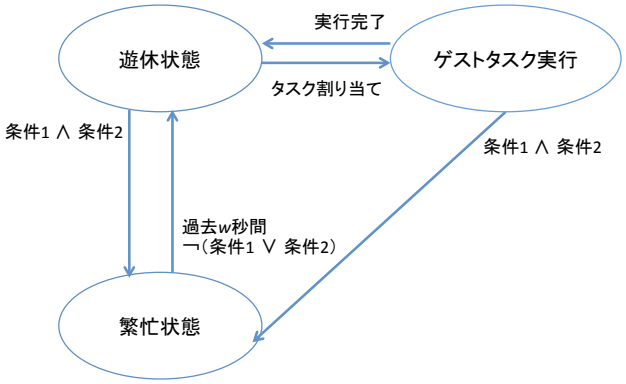


図 4 資源監視プロセスの状態遷移図

3.2 資源監視の手順

遊休計算資源を検出するために，提案システムでは各計算資源が起動した直後から資源監視プロセスを生成する．このプロセスは定期的に資源を監視し，条件 1. および条件 2. を満たすか否かを判定する．その状態遷移図を**図 4** に示す．遊休状態から繁忙状態へは，条件が成立すれば直ちに遷移する．一方，逆方向に関しては，遊休状態が w 秒間に渡り持続した場合にのみ遷移する．この検証時間はホストへの外乱を抑えるためにある．これにより，計算資源による頻繁なサーバ通信および，ゲストタスク中断を避ける．デフォルトの値は $w = 1$ である．

ある計算資源において状態遷移が発生すると，その計算資源は資源管理サーバへ自身の状態を通知する．その通知が遊休状態への遷移である場合，資源管理サーバはゲストタスクを該当する計算資源に投入する．投入されたゲストタスクは，ホスト上で協調マルチタスキングによりホストタスクとともに並行実行される．ゲストタスク実行中に，その計算資源が繁忙状態と判定された場合には，実行中のタスクを棄却し，その旨をサーバへ通知する．なお，現在のシステムはチェックポイントリスタートなどのゲストタスク再開機構は備えていない．

4. 評価実験

提案システムを評価するために，2 種類の実験を遂行した．**表 1** に実験環境を示す．

まず，協調マルチタスキングを評価するために，複数のホストアプリケーションを用意し，ゲストタスクと並行実行することにより，ホストに与えた外乱およびゲストに提供した性能を評価した．ゲストタスクとして CUDA SDK[6] の行列積を用い，協調マルチタスキングを実現できるように，そのプログラムを修正した．

次に，資源監視手法を評価するために，研究室環境において 4 台の計算機を用意し，1 ヶ月に渡り運用した．これ

表 1 実験環境

OS	Windows 7 Professional 64 bit
CPU	Intel Core i7 3770K (3.5 GHz)
メモリ	16 GB
GPU	NVIDIA GTX 680
CUDA バージョン	5.0
ドライババージョン	310.90

表 2 各状態のフレームレート（fps）

ホストタスク	専有実行	協調マルチタスキング
YouTube	28.7	28.6
OpenGL	60.0	60.0

らの計算機は研究室の学生が日常的に使用していて，主な用途はプログラム開発，文書作成およびウェブ閲覧である．運用時に得られた実行履歴を基に既存システム [3] と比較した．比較項目は，検出した遊休時間の総和，ゲストタスクのスループット，状態遷移によるサーバへの通知回数である．

4.1 協調マルチタスキングの評価

協調マルチタスキングに起因する外乱の大きさを確認するために，ホストタスクとして以下の 3 種を用意し，行列積と並行実行した．

S1. なし（ゲストタスクの専有状態）
S2. YouTube
S3. OpenGL レンダラ [7]

YouTube は特定の動画を再生するものであり，OpenGL レンダラは Phong シェーディングを反復するものである．なお，行列のサイズは 3072×3072 であり，行列積を 100 回繰り返した．

表 2 に計測したフレームレートを示す．協調マルチタスキングにより，ゲストタスクの実行時においても両者のフレームレートをほぼ維持できており，ホストへの外乱が抑えられている．なお，S1 に関しては，画面の更新そのものが発生せず，外乱が生じえない．したがって，フレームレートを測定していない．

表 3 に行列積のスループット比 P および実行モードの比 Q を示す．なお，$P = t_1/t_2$ であり，t_1 は行列積を専有実行した場合の実行時間を表し，t_2 は協調マルチタスキング時の行列積の実行時間を表す．また，$Q = q_1/(q_1 + q_2)$ であり，q_1 および q_2 はそれぞれ連続実行モードを選択した回数および定期実行モードを選択した回数を表す．ホストタスクを実行しない場合，ゲストタスクは計算資源を専有でき，実際に提案システムは約 93%のサブタスクを連続実行モードで実行できた．結果，スループット比 P の低下を約 6% に抑えられた．一方，YouTube および OpenGL レンダラは GPU の計算負荷を高めていて，前者では約 78%，後者ではすべての実行において定期実行モードを選択した．

表 3 各状態におけるスループット比および実行モード比

ホストタスク	スループット比 P（%）	実行モード比 Q（%）
なし	93.7	92.8
YouTube	45.9	21.8
OpenGL	35.0	0.0

表 4 各計算機の総稼働時間（時間）

	PC1	PC2	PC3	PC4
総稼働時間	135.1	15.9	197.0	81.6

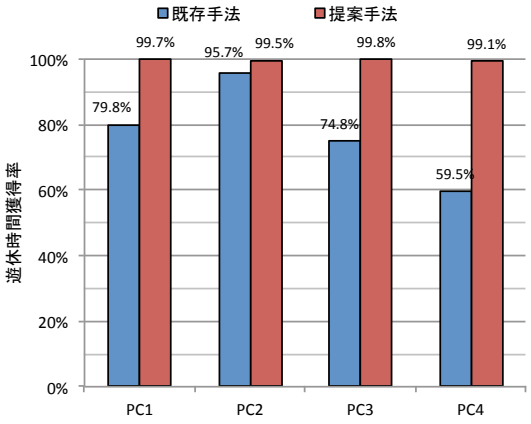


図 5 稼働時間に対する遊休時間の占める割合

その結果，スループット比が半分以下に低下している．

4.2 資源監視手法の評価

4 台の計算機 PC1〜PC4 を提案システムの計算資源として登録し，その実行履歴を取得した．なお，今回の実験は計算資源単体での実験であり，資源管理サーバは用意していない．各計算資源上の資源監視プロセスは，遊休状態を検出するたびにゲストタスクを実行するものとし，実行完了後にすぐさまゲストタスク実行を繰り返す．なお，1 個のゲストタスクは 3072×3072 の行列積を 50 回繰り返す．また，3.1 節で導入したパラメータ x，y および w は，提案手法および既存手法ともに同じ値を用いた．**表 4** に各計算機の総稼働時間を示す．

まず各手法が検出できた遊休時間を評価する．**図 5** に，総稼働時間に対する遊休時間の割合を示す．提案手法が検出した遊休時間は，すべての PC において長くなり，既存手法に対して 4%〜40%向上している．提案手法では，遊休時間が 99%以上を占めていることから，計算資源を酷使しない対話的な作業を行なっていたことがわかる．具体的には，文書作成やウェブ閲覧などが該当する．これらの作業はマウス・キーボードの入力を伴うものであり，既存手法では繁忙状態とみなされていた．したがって，マウス・キーボード入力を監視しない提案手法では遊休状態とみなせ，より多くの遊休時間を検出できた．

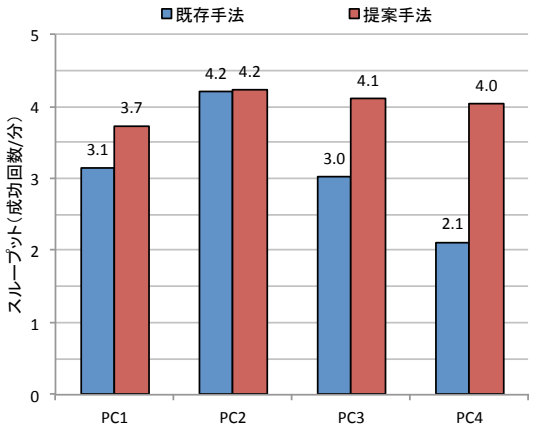


図 6 ゲストタスクのスループット

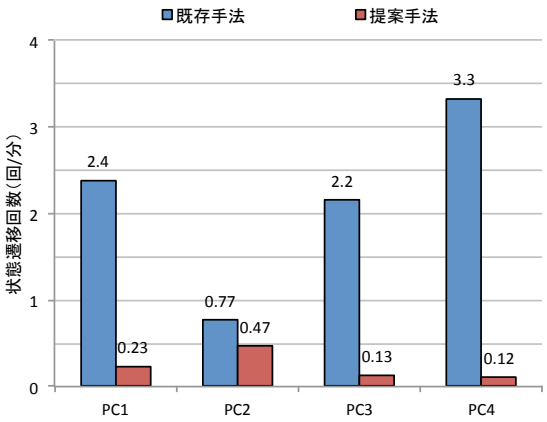


図 7 1 分あたりの状態遷移回数

表 5 タスク成功率（%）

	既存手法	提案手法
PC1	56.8	94.2
PC2	84.5	90.3
PC3	58.3	96.9
PC4	38.8	97.3

次に，ゲストタスクのスループットを評価した．**図 6** に，1 分あたりのタスク成功回数を示す．既存手法と比較して，提案手法は最大 2 倍のスループットを達成できた．協調マルチタスキングによる並行実行は，専有実行に対してスループットが低下する．しかし，検出できた遊休時間が増加したため，スループットが向上した．

表 5 にタスク成功率を示す．提案手法は，繁忙状態への遷移が少ないため，成功率が最大で 40%向上した．このことから，無駄なゲストタスク実行を抑えられることがわかる．以上のことからタスクの実行性能は向上したと言える．

最後に，サーバとの通信回数を評価するために，発生した状態遷移の回数を計測した．**図 7** に 1 分あたりの状態遷移回数を示す．既存手法と比較して，提案手法は 40%〜96%の状態遷移回数を削減できた．したがって，サーバへ同時に接続できるクライアント数を増加できる．

5. まとめと今後の課題

本稿では，ミリ秒単位の遊休時間を活用するために，協調マルチタスキングを用いた GPU グリッドシステムを提案した．協調マルチタスキングはホストタスクおよびゲストタスクの並行実行を実現し，ホストへの外乱，すなわちフレームレートの低下を抑えることができる．提案システムの資源監視プロセスは，マウス・キーボード入力の替わりに CPU および GPU の使用率を監視し，状態遷移をサーバへ通知する．これによりサーバとの通信回数を削減できる．

実験の結果，協調マルチタスキングはフレームレートを維持しつつゲストタスクを実行でき，性能低下を 35%に抑えた．また，資源監視手法に関しては，研究室内の学生が使用する 4 台の計算機を用いて 1 ヶ月に渡り運用した．既存手法と比較して，提案手法は検出できた遊休時間が最大で 40%向上し，最大 2 倍のスループットを達成できた．また，最大 96%の状態遷移回数を削減し，サーバの圧迫を抑えることができた．

今後の課題として，ゲストタスクに実用的なプログラムを用いた評価や，大規模環境での運用が挙げられる．

謝辞 本研究の一部は，JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」，科研費 23300007，および 23700057 の補助による．

参考文献

[1] NVIDIA Corporation: CUDA Programming Guide Version 4.2 (2012).
[2] The Folding@Home Project: Folding@Home Distributed Computing (2010). http://folding.stanford.edu/.
[3] Ino, F., Munekawa, Y. and Hagihara, K.: Sequence Homology Search Using Fine Grained Cycle Sharing of Idle GPUs, *IEEE Trans. Parallel and Distributed Systems*, Vol. 23, No. 4, pp. 751–759 (2012).
[4] 老田健太郎，伊野文彦，萩原兼一：文書作成および科学計算を両立する GPU 向け協調マルチタスキングの検討，第 10 回ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集（HPCS 2010），p. 63 (2010).
[5] Microsoft Corporation: Windows API (2009). http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx.
[6] NVIDIA Corporation: GPU Computing SDK (2012). http://developer.nvidia.com/gpu-computing-sdk/.
[7] Teranishi, T.: Phong shading sample program (2003). http://www.asahi-net.or.jp/~yw3t-trns/opengl/samples/fshphong/.